Figure 1

Modified SSA-conversion process

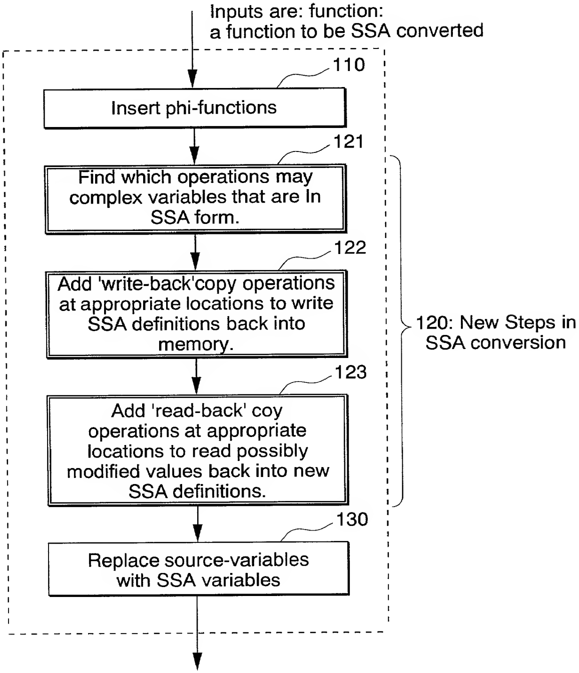

Inputs are: function:
a function to be SSA converted
110
Insert phi-functions
121
Find which operations may complex variables that are In SSA form.
122
Add 'write-back'copy operations at appropriate locations to write SSA definitions back into memory.
123
Add 'read-back' coy operations at appropriate locations to read possibly modified values back into new SSA definitions.
120: New Steps in SSA conversion
130
Replace source-variables with SSA variables

Figure 2

Overall compiler control flow

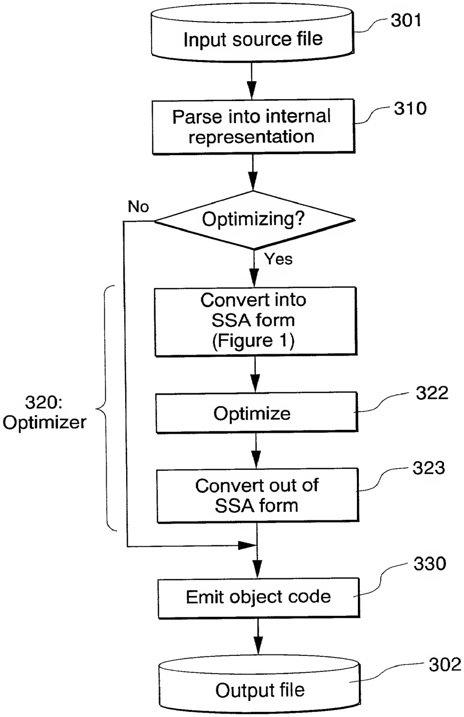


Figure 3

Program representation

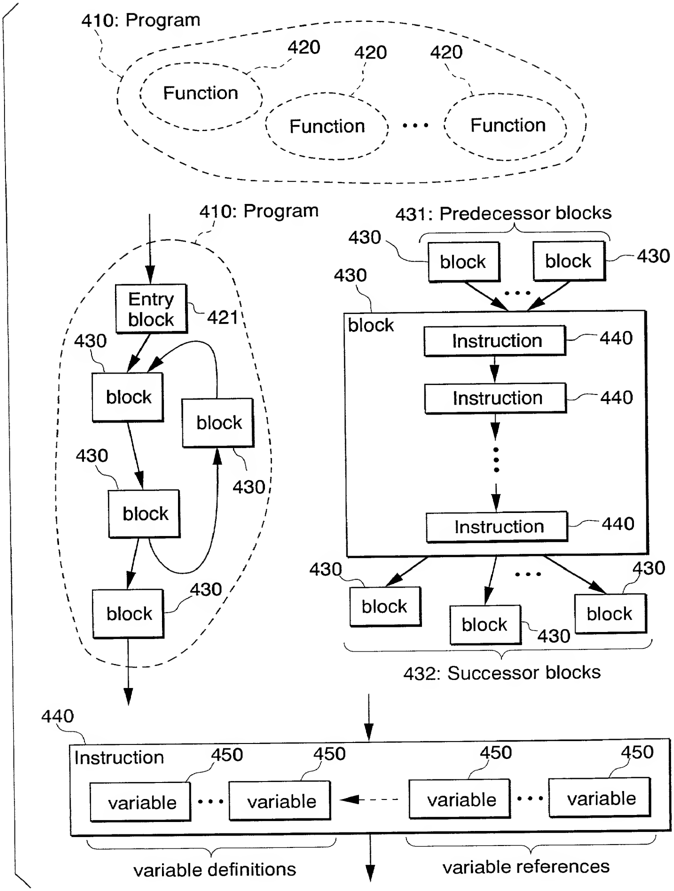


Figure 4

Placement of read/write-backs for the SSA form of *x, (*x)'1

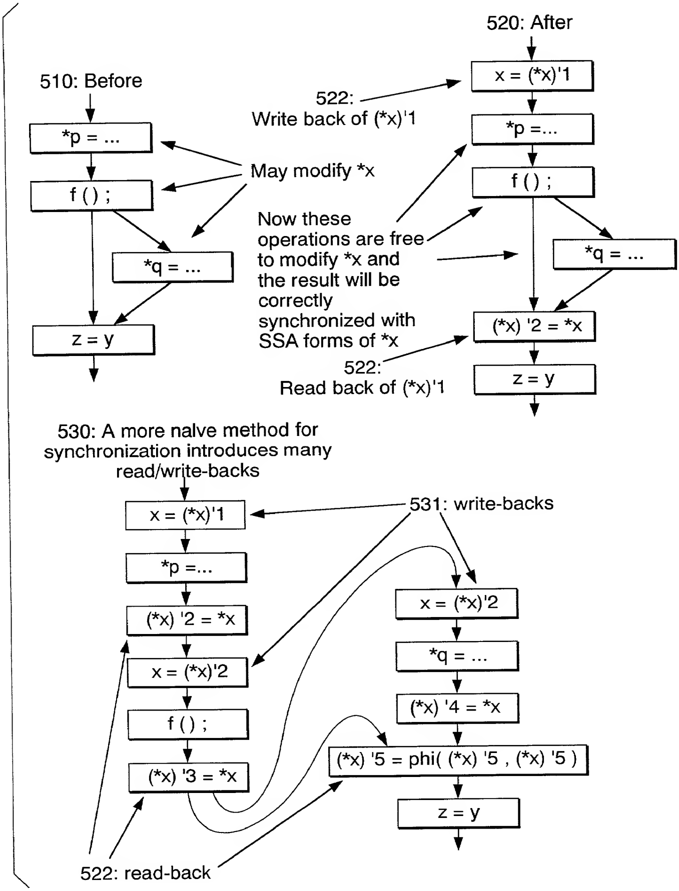


Figure 5

The procedure 'add_syncs_and_write_backs'

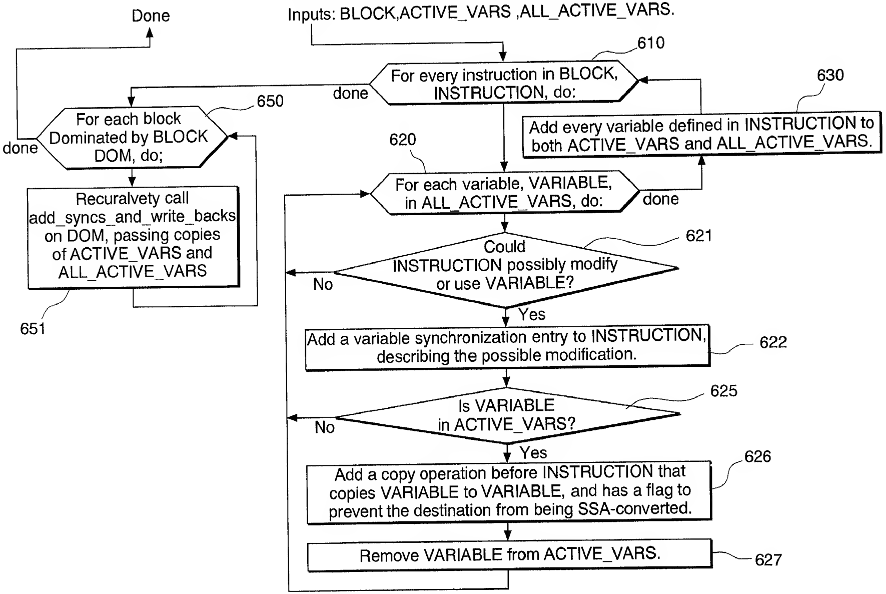

Inputs: BLOCK,ACTIVE_VARS ,ALL_ACTIVE_VARS.
610
For every instruction in BLOCK, INSTRUCTION, do:
done
630
Add every variable defined in INSTRUCTION to both ACTIVE_VARS and ALL_ACTIVE_VARS.
620
For each variable, VARIABLE, in ALL_ACTIVE_VARS, do:
done
621
Could INSTRUCTION possibly modify or use VARIABLE?
No
Yes
Add a variable synchronization entry to INSTRUCTION, describing the possible modification.
622
625
Is VARIABLE in ACTIVE_VARS?
No
Yes
626
Add a copy operation before INSTRUCTION that copies VARIABLE to VARIABLE, and has a flag to prevent the destination from being SSA-converted.
Remove VARIABLE from ACTIVE_VARS.
627
Done
650
For each block Dominated by BLOCK DOM, do;
done
Recuralvety call add_syncs_and_write_backs on DOM, passing copies of ACTIVE_VARS and ALL_ACTIVE_VARS
651

Figure 6

Conversion step (a'.III), insertion of read-backs

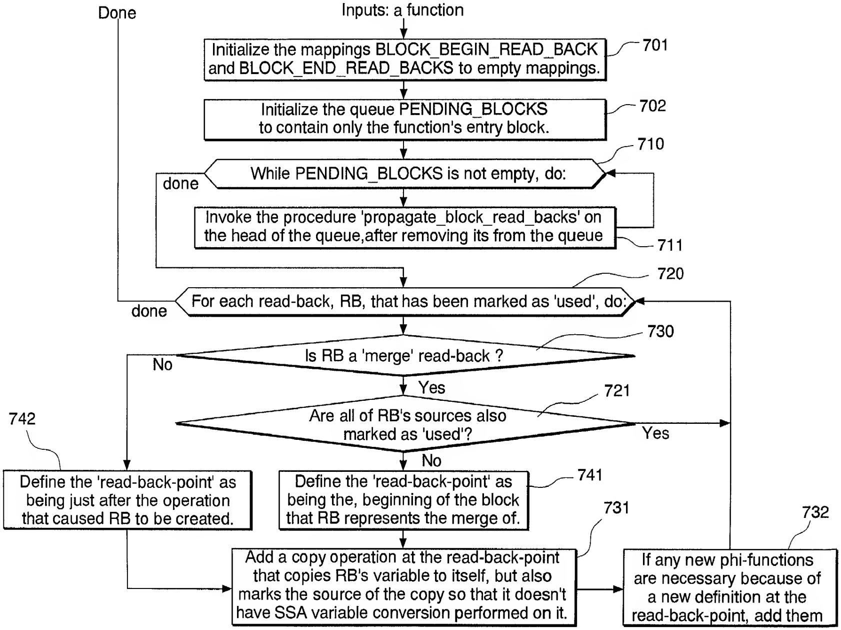

Inputs: a function
Done
Initialize the mappings BLOCK_BEGIN_READ_BACK and BLOCK_END_READ_BACKS to empty mappings.
701
Initialize the queue PENDING_BLOCKS to contain only the function's entry block.
702
710
While PENDING_BLOCKS is not empty, do:
done
Invoke the procedure 'propagate_block_read_backs' on the head of the queue,after removing its from the queue
711
720
For each read-back, RB, that has been marked as 'used', do:
done
730
Is RB a 'merge' read-back ?
No
Yes
721
Are all of RB's sources also marked as 'used'?
Yes
No
742
Define the 'read-back-point' as being just after the operation that caused RB to be created.
Define the 'read-back-point' as being the, beginning of the block that RB represents the merge of.
741
731
732
Add a copy operation at the read-back-point that copies RB's variable to itself, but also marks the source of the copy so that it doesn't have SSA variable conversion performed on it.
If any new phi-functions are necessary because of a new definition at the read-back-point, add them

Figure 7A

The procedure 'propagate_read_backs'

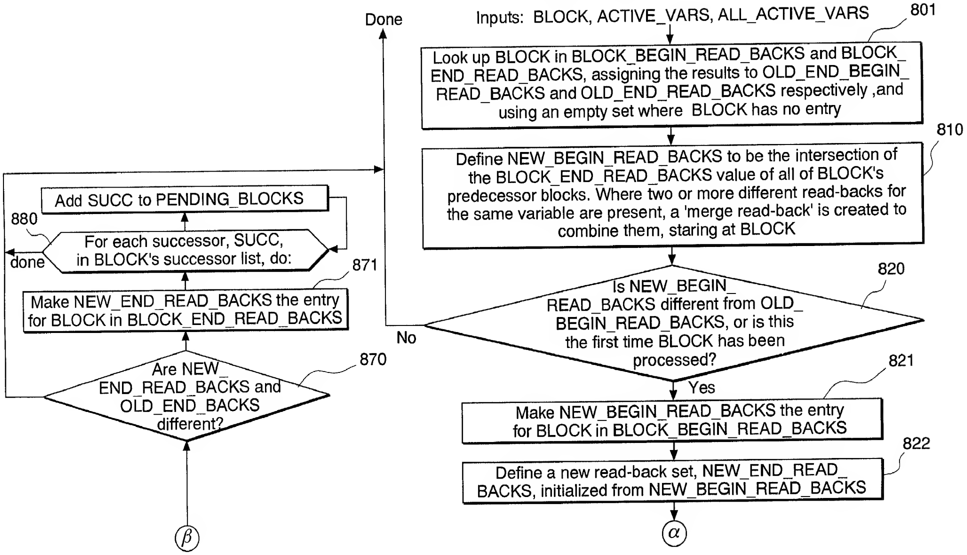

Inputs: BLOCK, ACTIVE_VARS, ALL_ACTIVE_VARS
801
Look up BLOCK in BLOCK_BEGIN_READ_BACKS and BLOCK_END_READ_BACKS, assigning the results to OLD_END_BEGIN_READ_BACKS and OLD_END_READ_BACKS respectively ,and using an empty set where BLOCK has no entry
810
Define NEW_BEGIN_READ_BACKS to be the intersection of the BLOCK_END_READ_BACKS value of all of BLOCK's predecessor blocks. Where two or more different read-backs for the same variable are present, a 'merge read-back' is created to combine them, staring at BLOCK
820
Is NEW_BEGIN_READ_BACKS different from OLD_BEGIN_READ_BACKS, or is this the first time BLOCK has been processed?
No
Done
Yes
821
Make NEW_BEGIN_READ_BACKS the entry for BLOCK in BLOCK_BEGIN_READ_BACKS
822
Define a new read-back set, NEW_END_READ_BACKS, initialized from NEW_BEGIN_READ_BACKS
α
β
870
Are NEW_END_READ_BACKS and OLD_END_BACKS different?
871
Make NEW_END_READ_BACKS the entry for BLOCK in BLOCK_END_READ_BACKS
880
For each successor, SUCC, in BLOCK's successor list, do:
done
Add SUCC to PENDING_BLOCKS

Figure 7B

The procedure 'propagate_read_backs'

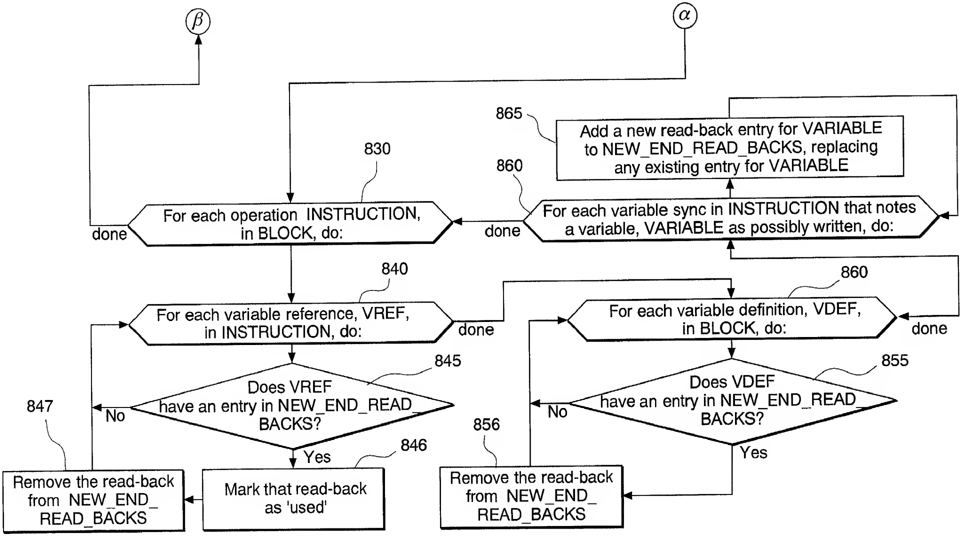

β
α
865
Add a new read-back entry for VARIABLE to NEW_END_READ_BACKS, replacing any existing entry for VARIABLE
830
860
For each operation INSTRUCTION, in BLOCK, do:
done
done
For each variable sync in INSTRUCTION that notes a variable, VARIABLE as possibly written, do:
840
860
For each variable reference, VREF, in INSTRUCTION, do:
done
For each variable definition, VDEF, in BLOCK, do:
done
845
855
847
Does VREF have an entry in NEW_END_READ_BACKS?
No
Yes
846
856
No
Does VDEF have an entry in NEW_END_READ_BACKS?
Yes
Remove the read-back from NEW_END_READ_BACKS
Mark that read-back as 'used'
Remove the read-back from NEW_END_READ_BACKS

Figure 8
Example source program

This short C program is used to illustrate the invention:

```
extern int g () , h ( ) , i ( ) , x;
int foo (int *p)
{
   (*p) ++;                                    [810]
   if (*P > 10)
      {
         g ( ) ;
         h () ;
         if (x > 5 )
            g () ;
         if (x > 3)
            i ( );
         else
            X = *p;
         *P = 5;
      }
   return *p;
}
```

Here's the same program converted to a slightly more primitive form:

```
int foo (int *p)
{
block1:
   *p := *p +1;                                [B20]
   if (*P <= 10)
      goto block8;
block2:
   g () ;
   h () ;
   if (x <= 5)
      goto block4;
block3:
   g () ;
block4:
if (x > 3)
   goto block6;
block5 :
   x :=  *p;                                   [B40]
   goto block7;
block6 :
   i ();
block7 :
   *p := 5;                                    [830]
block8:
   return *p;
}
```

Figure 9

SSA converted program, with simple implementation of read-backs :

The following is psuedo-C, augmented with the ` phi' operation, where

```
RESULT = phi (block1: VAL1 , ...,  blockN:VALN)
```

means ` assign VAL1 to RESULT if control-flow comes from block1 ' , and similarly so on for each value of N.

The extra variables 'pvN", where N is an integer, are SSA versions of *P, and are in fact local variables, not dereferences of p.

```
1nt foo (int *p)
{
  int pvl, pv2 , pv3 , pv4, pv5, pv6;

block1;
  pvl = *P + 1;
  if (pvl <= 10)
    goto block8;
block2:
  *P = pvl;          /* This writes-back PV1 to *P. */
  g ();
  pv2 = *P;          /* This reads-back *P into PV2. */
  *P = pv2;          /* This writes-back PV2 to *P. */
  h () ;
  pv3 = *P;          /* This reads-back *P into PV3 */            [912]
  if (x <= 5)
    goto block4;
block3:
  *p = pv3;          /* This writes-back PV4 to *p, */
  g () ;
  pv4= *p;           /* This reads-back *p into PV4. */           [911]
block4:
  pv5 = phi (block3: pv4, block2: pv3)                            [910]
  if (x > 3)
    goto block6;
block5:
  gota block7;
block6;
  i () ;
block7:
  x = phi (block6: x, block5: pv5);
block8:
  pv6 = phi (block1: pv1,  block7: 5);
  *P = pv6;          /* This writes-back PV6 to *P. */
  return pv6;
}
```

Figure 10

SSA converted program,
with the implementation of read-backs described in this patent

```
int foo (int *p)
{
  int pv1, pv2, pv3;

block1:
  pv1 = *p +1;                                                        [1011]
  if (pvl <= 10)
    goto block8;

block2;
  *p = pv1;          /* This writes-back pv1 to *P. */
  g ( ) ;                                                             [1021]
  h ( ) ;                                                             [1022]
  if (x <= 5)
    goto block4;

block3;
  g ();                                                               [1023]

block4:
  pv2 = *p;          /* This reads-back *p into pv2, */               [1030]
  if (x > 3)
    goto block6;

block5;
  goto block7;

block6:
  i ( );                                                              [1024]

black7 :
  x = phi (block6 : x, block5 : pv2) ;                                [1031]

block8:
  pv3 = phi (block1: pv1, block7: 5);                                 [1010]
  *P = pv3;           /*This writes-back PV3 to *P */

  return pv3;
}
```

Figure 11

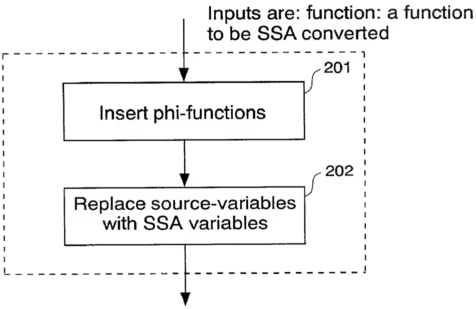
Register-alloced and SSA-unconverted program

using BBA-form requires having a good register allocator that will merge variables where possible, as it tends to generate a lot of variables with short lifetimes. We assume that here.

```
int foo (int *p)
{
   int pv;

block1;
   pv = *p + 1;
   if (pv <= 10)
      goto block8;

block2:
   'P = pv;            /* This writes-back pv to *P. */
   g () ;
   h () ;
   if (x <= 5)
      goto block4;

block3 :
   g () ;

block4 :
   if (x > 3)
      goto block6;

block5:
   x=*p;
      goto block7;

block6:
   i ();

block7:
    pv =5;

block8:
   *P= pv            /* This writes-back PV to *P. */

return pv;
}
```

Figure 12

Original SSA-conversion process


Inputs are: function: a function to be SSA converted
201
Insert phi-functions
202
Replace source-variables with SSA variables